Proceedings Intern. AMSE Conference Neural Networks, San Diego (USA), May 29-31, 1991, Vol. 1, pp. 75-86

Implementing Back Propagation Neural Nets with Logarithmic Arithmetic

by

M. Arnold, T. Bailey, J. Cowles, and J. Cupal

University of Wyoming

1 Introduction

The back propagation training algorithm [18] has been used successfully in a variety of applications where supervised learning is appropriate. After the training phase, the learned data is recalled by a large series of simple computations where each processor (neuron) computes a non-linear function of a weighted sum of products. In a layered feed forward network, the output from processor j on layer L is

$$x_j^L = f\left(\sum_{i=1}^{N_L} w_{i,j}^L \cdot x_i^{L-1} + \theta_j^L\right) \tag{1}$$

where x^{L-1} is a vector of inputs to processor j on layer L (the outputs of the processors on layer $L-1$), w^L is a matrix of weights describing the connection strengths between layer $L-1$ and L, θ_j^L is an offset for unit j, and N_L is the number of units on layer L.

Any of several types of non-linearities might be used for the activation function, but the sigmoid logistic

$$f(z) = \frac{1}{1+e^{-z}} \tag{2}$$

is often used because of its desirable properties. It is a continuous differentiable monotone increasing function with $\lim_{z\to-\infty} f(z) = 0$, $f(0) = 0.5$ and $\lim_{z\to\infty} f(z) = 1$, which means the output of each neuron can be represented with a number between zero and one. During training, errors in the output are propagated backwards to adjust the weights in a computation involving $f'(z)$. The majority of the computational resources in a neural network are used to obtain the sum of products, but a noticeable portion of the resources are required to compute the non-linearity.

This paper describes the use of logarithmic arithmetic to implement a back propagation neural network during both training and recall phases. Section 2 gives a review of the literature that indicates neural nets require only limited precision arithmetic. Section 3 describes logarithmic arithmetic, which has been shown to be more economical than digital fixed point or floating point arithmetics in digital signal processing applications that, like neural nets, require only limited precision. Section 4 explains two novel approximations of (2) which can be computed with logarithmic arithmetic at almost no cost. Section 5 gives simulation results that show limited precision logarithmic arithmetic with our proposed approximations of (2) converges in a number of iterations comparable to other more expensive techniques. Section 6 describes our conclusions.

2 Review of Literature

Opinions vary on the proper technologies for implementing neural nets. One issue is analog versus digital. Another consideration is training. A third design decision, which is interrelated to the others, is precision.

Some people argue that analog circuits are better since they are more *analogous* to biological neurons and can produce a sum of products more cost effectively than traditional digital hardware [3]. Despite this, back propagation is challenging to implement with purely analog circuits. For example, Mueller et. al. [16] decided on a hybrid approach, with analog feed forward computations, but digital back propagation.

Many favor purely digital implementations because they offer much greater flexibility in training and because they can be easier to interface to and/or be implemented with existing ("off the shelf") technology [25]. Digital implementations make the benefits of neural net research accessible to a wider audience, as evidenced by the widespread application of back propagation to a variety of problems.

For most neural net architectures, including back propagation, the precision required in any individual computation is small [2,8]. This is why analog circuits work at all, despite noise and the fact that it is hard to fabricate accurate analog components in VLSI. From a digital perspective, this means that the theoretical number of bits required to represent neuron activation and weights is much smaller than the 32 bit floating point implementation that is often employed. For example, Brown et. al. have shown [4] that an 8 bit floating point representation is adequate in some cases when the allocation of these eight bits between the exponent and mantissa is allowed to be programmable, instead of being static as is traditionally the case for larger floating point formats.

Many have used a fixed point binary representation. For example, 16 bit weights and 8 bit input/outputs are useful with many training algorithms [2,7,5]. Kamph et. al. [10] have shown that 5 bits is reasonable in a network with 64 neurons, and that an additional bit allows the network size to double. Siu and Bruck [20] have shown in theory that many computations involving n input neurons can be performed by neurons whose precision is $O(\log n)$. Marchesi et. al. [13] describe a modification of the back propagation algorithm where most weights become powers of two by the end of the training phase, but are continous during training. Shoemaker et. al. [19] have shown that back propagation can converge even when weight changes are quantized to one or two bits, but the back propagated δs must be computed with higher precision.

Others have exploited alternate arithmetic methods such as quasi serial arithmetic [25] and residue arithmetic [14].

3 Logarithmic Arithmetic

A different type of digital arithmetic, known as the sign logarithm number system, [22,24] offers a promising alternative to analog, fixed point or floating point methods of implementing back propagation nets. The sign logarithm number system uses a signed fixed point binary word containing the logarithm of the absolute value of the input or weight. An extra bit indicates the sign of the number being represented. The base of the logarithm, b, is arbitrary. The quantized logarithmic representation, x, together with the sign bit, x_s, represent a real number, y, as

$$x = \begin{cases} (2^M - 1)\cdot 2^{-F} & \text{if } \log_b(|y|) > (2^M - 1)\cdot 2^{-F} \\ \text{round}(2^F \cdot \log_b(|y|))\cdot 2^{-F} & \text{if } y \neq 0 \text{ and } |\log_b(|y|)| \leq (2^M - 1)\cdot 2^{-F} \\ -(2^M - 1)\cdot 2^{-F} & \text{if } y = 0 \text{ or } \log_b(|y|) < -(2^M - 1)\cdot 2^{-F} \end{cases} \quad (3)$$

$$x_s = \begin{cases} 0 & \text{if } y \geq 0 \\ 1 & \text{if } y < 0 \end{cases} \quad (4)$$

where F determines the number of bits of precision, M is $W - 2$, $R = M - F$ is the number of bits for the dynamic range, and W is the total number of bits in the word, including the sign bit (x_s) of the represented value (y) and the sign bit of x, which indicates if $|y| < 1.0$ or $|y| \geq 1.0$. The smallest positive number that can be represented is $b^{-(2^M-1)\cdot 2^{-F}}$ and the largest is $b^{(2^M-1)\cdot 2^{-F}}$. For example, with $b = e$, $M = 10$, $W = 12$ and $F = 8$, the smallest positive number that can be represented is $+0.018$ ($x = -1023 = -11.11111111_2$, $x_s = 0$) and the largest is $+54.4$ ($x = +1023/256 = +11.11111111_2$, $x_s = 0$). For the same precision, but $W = 13$ and $M = 11$, the smallest is $+0.0000336$ ($x = -111.11111111_2$, $x_s = 0$) and the largest is about $+2969$ ($x = 111.11111111_2$, $x_s = 0$).

Since $\log_b(0) = -\infty$, there is no exact representation of zero, but rather zero (or values near zero) are rounded to the smallest positive number. Using the above examples, with $b = e$, $M = 10$, $W = 12$ and $F = 8$, zero is represented as $x = -11.11111111_2$, $x_s = 0$, and with $W = 13$ and $M = 11$, zero is represented as $x = -111.11111111_2$, $x_s = 0$. Underflow is a potential source of problems. The justification for omitting zero is to simplify the hardware. Omitting a representation of zero is quite common in logarithmic arithmetic implementations [6].

Multiplications are implemented by low cost fixed point additions, and are completely accurate. Computing the sum of two values with the same sign, z_1 and z_2, given $\log_b|z_1|$ and $\log_b|z_2|$ uses

$$\log_b(|z_1| + |z_2|) = \log_b|z_1| + s_b(\log_b|z_2| - \log_b|z_1|) \quad (5)$$

where s_b is the *addition logarithm* given by $s_b(z) = \log_b(1 + b^z)$. When z_1 and z_2 have different signs, computing the sum uses

$$\log_b(|z_1| - |z_2|) = \log_b|z_1| + d_b(\log_b|z_2| - \log_b|z_1|) \quad (6)$$

where d_b is the *subtraction logarithm* given by $d_b(z) = \log_b|1 - b^z|$, and the sign of the result is sign of the larger of z_1 or z_2.

Because the precision required in a neural network is limited, these functions can be implemented using a table lookup. If N_1 is the number of nodes in the input layer, N_2 is the number of nodes in the hidden layer, N_3 is the number of nodes in the output layer and M is described above, the number of bits required for the weights is $M \cdot N_2 \cdot (N_1 + N_3)$ while the number of bits required for the s_b and d_b tables is $M \cdot 2^{M+1}$. When M is small and N_1, N_2 and N_3 are large, the amount of space used by the table is insignificant compared to space used by the weights. For example, if $M = 10$ and $N_1 = N_2 = N_3 = 100$, the table requires 20,480 bits, while the weights require 200,000 bits. Additional techniques [1,21,12] have been described which can further reduce the size of the tables.

Logarithmic arithmetic has been used successfully in digital signal processing applications that have precision requirements similar to neural nets, including the FFT [23], hidden Markov speech recognition [17], and very low power FIR filters for wearable hearing aids [15]. Lang et. al. [12] fabricated similar VLSI multiply accumulate circuits using conventional and logarithmic designs and found the logarithmic design is better than conventional designs when the precision required is moderate, as could be the case for a neural net.

To our knowledge, no previous digital implementation of any kind of neural net has used logarithmic arithmetic. On the other hand, analog logarithmic representations have been used in some aspects of neural circuitry, such as representation of weights [16]. Furthermore, the nearly logarithmic response of human visual and auditory systems argues for the appropriateness of logarithmic representation [6].

4 Sigmoid Approximation

The combination of back propagation with logarithmic arithmetic has an advantage that has not previously been reported in the literature: When $b = e$, the derivative of the addition logarithm is the sigmoid logistic function, $f(z) = s_e'(z)$ and $\log(f(-z))$ is the addition logarithm. For a conventional fixed or floating point processor, a table of the non-linearity would have to be stored in memory. By exploiting properties of the addition logarithm described here, approximations to $f(z)$ and $f'(z)$ can be computed by a sign logarithm processor in a single cycle without the additional table required by a fixed or floating point processor. This makes the cost of implementing back propagation with logarithmic arithmetic quite attractive. The size of the memory that implements both the sigmoid and logarithmic addition/subtraction is no bigger than the sigmoid table that would be required in a conventional implementation.

Assume z is a *fixed point* value, and x is the *sign logarithm* equivalent of z, which is $\log_e|z|$. The sign of z must be given separately in sign logarithm format as x_s, which is 1 if z is negative and 0 if z is positive. We want to compute the logarithm of the sigmoid in preparation for future computations. Note that the output from the sigmoid is always positive, so there is no need to have a sign bit on the output.

There are four cases to consider. The first, when $z \geq 1$, is indicated by $x_s = 0$ and $x \geq 0$.

This case is computed as

$$\log_e(f(z)) = \log_e\left(\frac{1}{1+e^{-z}}\right) = \log_e\left(\frac{e^z}{e^z+1}\right) = z - \log_e(e^z+1) = z - s_e(z) = |z| - s_e(|z|). \quad (7)$$

The second case occurs when $0 < z < 1$. This case is indicated by $x_s = 0$ and $x < 0$. In this region, the sigmoid can be accurately interpolated using the straight line

$$f(z) \approx [f(1) - f(0)] \cdot z + f(0). \quad (8)$$

This computation can be performed in sign logarithm arithmetic as

$$\log(f(z)) \approx c_1 + s_e(c_2 + x) \quad (9)$$

The third case is when $-1 < z < 0$, indicated by $x_s = 1$ and $x < 0$. Also in this region, the sigmoid can be approximated by (8), but the sign logarithm arithmetic has to account for the fact that $z < 0$:

$$\log_e(f(z)) \approx c_1 + d_e(c_2 + x) = c_1 + c_2 + d_e(-(c_2 + x)) \quad (10)$$

The last case is when $z \leq -1$, indicated by $x_s = 1$ and $x > 0$. Here

$$\log_e(f(z)) = \log_e(f(-|z|)) = \log_e\left(\frac{1}{1+e^{|z|}}\right) = -s_e(|z|). \quad (11)$$

Unfortunately, sign logarithm hardware does not provide $|z|$ in fixed point form, as is required in the first and fourth cases. Instead, it provides $x = \log_e|z|$, which is directly usable only in the second and third cases. Therefore, to compute the non-linearity for the first and fourth cases it is necessary to approximate $|z| = e^x$ using fixed point arithmetic. One crude approximation that is easy to compute comes from the first two terms of the Taylor series expansion for e^x: $|z| \approx 1 + x$, where $|z| \geq 1$. Another (slightly better) approximation is $|z| \approx 1 + 2x$ which is easy to compute in fixed point binary arithmetic by shifting x one place and adding one.

When the cases are combined with the first technique for computing e^x, the approximation for the logarithm of $f(z)$ is

$$f_1(x_s, x) = \begin{cases} 0 & \text{if } x_s = 0 \text{ and } 1 + x > (2^M - 1) \cdot 2^{-F} \\ (1+x) - s_e(1+x) & \text{if } x_s = 0,\ x \geq 0, \text{ and } 1 + x \leq (2^M - 1) \cdot 2^{-F} \\ c_1 + s_e(c_2 + x) & \text{if } x_s = 0 \text{ and } x < 0 \\ c_1 + d_e(c_2 + x) & \text{if } x_s = 1 \text{ and } x < 0 \\ -s_e(1+x) & \text{if } x_s = 1,\ x \geq 0, \text{ and } 1 + x \leq (2^M - 1) \cdot 2^{-F} \\ -(2^M - 1) \cdot 2^{-F} & \text{if } x_s = 1 \text{ and } 1 + x > (2^M - 1) \cdot 2^{-F}. \end{cases} \quad (12)$$

The first and last cases deal with overflow of $1 + x$. The maximum error is $6.639 \cdot 10^{-2}$.

Using the second approach for computing e^x, the approximation for the logarithm of $f(z)$

is

$$f_2(x_s, x) = \begin{cases} 0 & \text{if } x_s = 0 \text{ and } 1 + 2x > (2^M - 1) \cdot 2^{-F} \\ (1+2x) - s_e(1+2x) & \text{if } x_s = 0,\ x \geq 0, \text{ and } 1 + 2x \leq (2^M - 1) \cdot 2^{-F} \\ c_1 + s_e(c_2 + x) & \text{if } x_s = 0 \text{ and } x < 0 \\ c_1 + d_e(c_2 + x) & \text{if } x_s = 1 \text{ and } x < 0 \\ -s_e(1+2x) & \text{if } x_s = 1,\ x \geq 0, \text{ and } 1 + 2x \leq (2^M - 1) \cdot 2^{-F} \\ -(2^M - 1) \cdot 2^{-F} & \text{if } x_s = 1 \text{ and } 1 + 2x > (2^M - 1) \cdot 2^{-F}. \end{cases} \qquad (13)$$

The maximum error is $4.235 \cdot 10^{-2}$.

5 Simulations

We used three different sets of exemplers to simulate the impact of limited wordsize logarithmic arithmetic on the convergence of back propagation networks. The first set of exemplers is for the well known exclusive OR problem. The second set of exemplers is a binary encoder decoder. The third set of exemplers is oil well logging data from the Pitchfork and Oregon Basin fields near Meeteetse, Wyoming.

In every experiment, we used a layered feed forward network with one hidden layer. The training algorithm used in all experiments is simple back propagation. Enhancements such as momentum and noise were not used. During each epoch, each exempler was presented once, and the weights were updated after presentation of just that exempler. The training was allowed to proceed only up to a specified maximum, typically 100 epochs.

For every simulation, given a certain set of exemplers and a set of initial weights, we categorized the outcome into one of three classes: converged, stuck, or indeterminate. If all output(s) of the net came within a specified convergence criteria, typically 0.1, we classified it as converged. The stuck cases were those where, at the end of an epoch, the weights cycled back to *exactly* the same representation they had at the beginning of the epoch. The indeterminate cases simply failed to converge by the arbitrary maximum number of epochs we specified. The random initial weights were taken from a uniform distribution, typically $[-2, 2]$.

In many experiments, we ran both a 32 bit floating point (23 bits of precision) net as well as various precisions and wordsizes of logarithmic arithmetic nets. In the experiments involving logarithmic arithmetic, we used $f_1(x_s, x)$ and $f_2(x_s, x)$ as well as the closest approximation to $f(z)$ that is possible in the limited precision system, which we refer to as $f_0(x_s, x)$. We used f_0 for comparison purposes only, as it would be expensive to realize in hardware. In a few experiments we varied the learning rate, η, and the convergence criteria. Except for the well logging problem, the primary concern was to observe the effect of precision, range, and sigmoid approximation on the number of cases that converged, and on how rapidly those cases converged.

XOR Experiments

We conducted four experiments with the XOR problem. In the first experiment, we varied the sigmoid and the precision, F. (Varying F caused the wordsize, W to vary proportionately.) In the second experiment, we used f_2 and held the wordsize fixed, and varied the tradeoff between precision and range. In the third experiment, we varied the number of hidden units while using precision, range, and sigmoid that seemed reasonable from earlier experiments. In the final XOR experiment, we varied the convergence criteria and precision.

In the first experiment we held the range of the logarithmic number system constant at $R = 3$ bits, and varied the precision from $F = 7$ to $F = 10$ using all three sigmoid approximations. The maximum number of epochs allowed was 2000, convergence criteria was 0.1, and $\eta = 2.0$. For each combination of precision and sigmoid approximation, the same set of 100 initial random weights chosen from $[-2, 2]$ were used. For comparison, we used the same weights to train a 32 bit floating point version. The results are shown in Table 1.

The floating point version converged in 58 of the 100 cases, and logarithmic versions converged in a comparable number of cases (73 to 45) for $F \geq 9$. Although f_1 is slightly better than either f_2 or f_0 for $F \geq 9$, f_2 allows convergence of about as many cases with $F = 8$ as f_0 allows with $F = 9$. Also, there are no convergent cases using f_1 with $F = 8$. Since none of the $F = 7$ cases converged, we conclude that using f_2 with $W = 13$ is about as small as is possible with two hidden units and a convergence criteria of 0.1.

sigmoid	F	R	M	W	converged	stuck	indeterminate	arithmetic
f_2	10	3	13	15	69	29	2	logarithmic
f_1					73	20	7	
f_0					54	44	2	
f_2	9	3	12	14	54	45	1	
f_1					55	43	2	
f_0					45	54	1	
f_2	8	3	11	13	44	55	1	
f_1					0	97	3	
f_0					32	66	2	
f_2	7	3	10	12	0	98	2	
f_1					0	95	5	
f_0					0	96	4	
f	23	8	-	32	58	0	42	FP base 2

Table 1. Comparison of logarithmic and floating point arithmetic for 2 hidden unit XOR with convergence criteria of 0.1 and $\eta = 2$. W is the number of bits in the word, F is the number of bits of precision, and R is the number of bits used for the range.

The stuck cases increase as precision is decreased. Similar phenomena have been reported with 16 bit fixed point implementations of back propagation [2]. We attribute this to situations where some δ(s) are small and therefore cannot accurately influence weight changes. In the typical situation we observed, the limited precision arithmetic tries to make fine adjustments to reasonably large weights. Although a new representation is given to the weight, it overshoots the proper value because of quantization. Presentation of a later exempler attempts to adjust it back, but now quantization puts the weight back to its original value. This situation can occur even for nets close to convergence. (In fact, it is reasonable to expect the net to get stuck just before convergence since this is when δs get small.)

From Table 1, it is clear that the logarithmic system is as (or more) likely to converge as the much more expensive floating point system. Another question in this experiment is whether the logarithmic representation slowed convergence. As Table 2 shows, the average number of epochs to converge is *smaller* with the limited precision logarithmic system. This is consistent with similar results that Baker [2] obtained for back propagation with 16 bit fixed point arithmetic.

F	R	M	W	epochs	arithmetic
10	3	13	15	378.9	logarithmic
9	3	12	14	371.2	
8	3	11	13	328.1	
23	8	–	32	407.3	FP base 2

Table 2. Average number of epochs to converge on XOR with 2 hidden units using f_2.

The conclusion of our first experiment indicates that limited precision logarithmic arithmetic performs at least as well as floating point arithmetic on this problem. At least 8 bits of precision are required for 2 hidden unit XOR with a convergence criteria of 0.1 and $\eta = 2.0$. One way to obtain the required precision at lower cost would be to reduce the number of bits used for representing the range from $R = 3$ to $R = 2$. At first glance, this would appear to be harmless, since this reduces the maximum representable number from 2969 to 54, which is still larger than any number this XOR net needs. Unfortunately, the smallest positive number grows from about 0.00003 to 0.02, which is more than twice as large as the minimum precision, e^{-F}. Fortunately, logarithmic number systems have a property not commonly found with floating point systems: the tradeoff between range and precision can be made continous. For example, with $W = 13$ it is no more difficult to implement $F = 8.12, R = 2.88$ than it is to implement $F = 8, R = 3$.

This enabled us to conduct a second experiment holding the word size constant and observing the effects of slight changes in range and precision. This was done with f_2 for $W = 13, M = 11$ (Figure 1) using two, three and six hidden units. For a 13 bit word with two hidden units, the optimal value is very near $F = 8$. The left edge of the curve is caused by inadequate precision and the sometimes more abrupt right edge of the curve is caused by limited range (too large a gap near zero). The initial weights and η are the same as the previous experiment.

In the third XOR experiment, we varied the number of hidden units from 2 to 32 with $F = 9, R = 3$. (Figure 2) The use of additional units was previously shown by Chauvin [18] to decrease the number of epochs required for convergence. The straight line in the Figure is what Chauvin predicts. Our results exhibit a similar behavior.

Our last experiment with XOR investigated the effect of the convergence criteria on the number of weights that converged with f_2 and $R = 3$ (Figure 3). As expected, a relaxation of the convergence criteria allows more cases to converge. With a convergence criteria of 0.25, word sizes as small as 11 or 12 bits can converge.

Encoder/Decoder

Although XOR is useful for testing the effects of logarithmic arithmetic and sigmoid approximation, it is unlike the typical application of back propagation. For example, XOR has only two inputs and one output. A simple problem that has multiple inputs and outputs is the encoder/decoder problem. This was trained with 20 initial random weights selected from $[-2, 2]$ using eight inputs and five hidden units for $\eta = 0.5$, $\eta = 1.0$ and $\eta = 2.0$. Floating point and logarithmic versions (f_2, $R = 3$ with $F = 11, 10, 9,$ and 8) were used for comparisons as shown in Table 3. As with XOR, the logarithmic arithmetic performed as well as floating point arithmetic.

F	R	$\eta = 0.5$	$\eta = 1.0$	$\eta = 2.0$	arithmetic
11	3	371.9	186.6	106.5	logarithmic
10	3	375.5	187.3	106.2	
9	3	384.7	189.9	106.1	
8	3	406.1	195.1	108.3	
23	8	392.0	201.0	108.8	FP base 2

Table 3. Average number of epochs for Encoder/Decoder to converge with 5 hidden units.

Well Logging

The final experiment we conducted is an attempt to observe how limited precision logarithmic arithmetic works with realistic data. We obtained data on an oil well logging problem. Three parameters, natural gamma ray radioactivity, formation bulk density, and neutron porosity are to be used to predict a fourth parameter, acoustic (P-wave) sonic transit time. The data came from the Pitchfork and Oregon Basin fields in northwestern Wyoming. Using 436 exemplers from a Pitchfork well, we trained an f_2 logarithmic net ($M = 8$, $F = 5$) with 6 inputs, 6 hidden units and one output for 4 epochs with $\eta = 2.0$. The exemplers were scaled to be between 0.0 and 1.0, and three additional inputs were provided based on transformations of the original values. The initial random weights were from $[-2, 2]$. The resulting network was then used to calculate the output values for a second data set taken at the Oregon field. This data set consisted of 401 data points. The RMS error of the results, 4.93, compares favorably with results obtained by Iverson and Walker [9] using a combination of empirical relationships and physical properties. They obtained an RMS error of 5.60.

6 Conclusions

Our simulations show that a limited precision logarithmic number system is adequate for implementing many kinds of layered feed forward neural nets trained with back propagation. The percentage of initial random weights that converged, and the average number of epochs required for convergence were comparable to much more expensive floating point implementations.

In logarithmic arithmetic, multiplication is the least expensive operation, instead of being the most expensive, as it is in fixed and floating point implementations. Addition and subtraction require a table in a logarithmic implementation. The cost of using logarithmic arithmetic in back propagation is low because of the novel sigmoid approximation techniques presented here that use the addition/subtraction table. The size of the addition/subtraction table is no bigger than the size of the sigmoid table required in a conventional implementation of back propagation. Our simulations have shown that the sigmoid approximation has only a minor (and sometimes beneficial) impact on convergence.

Although we have only considered a purely digital implementation, it is worth noting that the hybrid system designed by Mueller [16] uses digital back propagation with a logarithmic representation of weights. Perhaps logarithmic arithmetic could speed the back propagation training of such analog circuits.

Logarithmic arithmetic is not restricted only to back propagation training using the sigmoid logistic. Other common activation functions, such as the soft limiter, can be implemented easily. The advantages of logarithmic arithmetic have been proven by several successful integrated circuits [12,15,17] for DSP applications having precision requirements similar to back propagation neural nets. It may be that the area/speed properties of low precision logarithmic arithmetic are sufficiently compelling [12] that even when a non-sigmoid activation function is used that the logarithmic implementation would be preferred. A general purpose neurocomputer needs to provide for many training algorithms, including the most popular: back propagation. Our simulations suggest that logarithmic arithmetic may be the most economical way to implement back propagation with digital technology. We feel that future research in this area is warranted.

References

[1] M. G. Arnold, T. A. Bailey, and J. R. Cowles, "Improved Accuracy for Logarithmic Addition in DSP Applications," *Proc. ICASSP.*, pp. 1714-1717, New York, 1988.

[2] T. Baker and D. Hammerstrom, "Characterization of artificial neural network algorithms," *IEEE International Symposium on Circuits and Systems*, vol. 1, pp. 78-81, 1989.

[3] S. Bibyk and M. Ismail, "Issues in Analog VLSI and MOS Techniques for Neural Computing," in C. Mead and M. Ismail, *Analog VLSI Implementation of Neural Systems*, Kluwer, Norwell, MA, 1989, pp. 103-133.

[4] H. K. Brown et. al., "A Neural Network Integrated Circuit Supporting Programmable Exponent and Mantissa," *Proceedings of the 1990 Custom Integrated Circuits Conference*, p. 26.3/1-26.3/4, 1990.

[5] M. Duranton and J. A. Sirat, "Learning on VLSI: a general purpose digital neurochip," *Philips Journal of Research*, vol. 45, no. 1, p. 1-17, 1990.

[6] A.D. Edgar and S.C. Lee, "FOCUS Microcomputer Number System," *Commun. ACM*, vol. 22, p. 166, 1979.

[7] S. Garth and D. Pike, "An integrated system for neural network simulations," *Computer Architecture News*, vol. 16, no. 1, pp. 37-44, March 1988.

[8] P. W. Hollis and J. J. Paulos, "The effects of precision constraints in a backpropagation learning network," *IJCNN: International Joint Conference on Neural Networks* , vol. 2, p. 625, 1989.

[9] W. P. Iverson and J. N. Walker, "Shear and Compressional Logs Derived From Nuclear Logs," University of Wyoming Preprint.

[10] F. Kamph et. al., "Optimization of a digital neuron design," *23rd Annual Simulation Conference*, pp. 73-80, Nashville, April 23-27, 1990.

[11] N.G. Kingsbury and P.J.W. Rayner, "Digital Filtering Using Logarithmic Arithmetic," *Electron. Lett.*, vol. 7, p. 56, 1971.

[12] J. H. Lang, C. A. Zukowski, R. O. LaMaire, C. H. An, "Integrated Circuit Logarithm Arithmetic Units," *IEEE Trans. Comput.*, vol. C-34, p. 475-483, 1985.

[13] M. Marchesi et. al., "Design of multi-layer neural networks with powers-of-two weights," *IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 2951-4, 1990.

[14] G. Martinelli and R. Perfetti, "RNS Neural Networks," *IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 2955-2958, 1990.

[15] R. E. Morley, Jr. , G. L. Engel, T. J. Sullivan, S. M. Natarajan, "VLSI based design of a battery-operated digital hearing aid," *Proc. ICASSP.*, pp. 2512-2515, New York, 1988.

[16] Paul Mueller et. al., "Design and Fabrication of VLSI Components for a General Purpose Analog Neural Computer," in C. Mead and M. Ismail, *Analog VLSI Implementation of Neural Systems*, Kluwer, Norwell, MA, 1989, pp. 135-169.

[17] H Murveit et. al., "A large-vocabulary real-time continuous-speech Recognition System," *Proc. ICASSP.*, pp. 789-792, Glasgow, 1989.

[18] D. E. Rumelhart, G. F. Hinton, R. J. Williams, "Learning Internal Representations With Error Propagation," in D. E. Rumelhart, J. E. McClelland *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press, 1986, vol. 1, pp. 345-355, 1986.

[19] P. A. Shoemaker et. al., "Back-propagation learning with course quantization of weight updates," *IJCNN: International Joint Confernece on Neural Networks*, vol. 1, pp. 573-576, 1990.

[20] K. Y. Siu and J. Bruck, "Neural Computations of Arithmetic Functions," *Proc. IEEE*, vol. 78, pp. 1669-1675, Oct. 1990.

[21] T. Stouraitis, "Logarithmic Number System Theory, Analysis, and Design," Ph. D. Dissertation, University of Florida, Gainesville, 1986.

[22] E.E. Swartzlander and A.G. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Trans. Comput.*, vol. C-24, p. 1238, 1975.

[23] E.E. Swartzlander, D. Chandra, T. Nagle, and S.A. Starks, "Sign/Logarithm Arithmetic for FFT Implementation," *IEEE Trans. Comput.*, vol. C-32, p. 526, 1983.

[24] F.J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20 Bit Logarithmic Number System Processor," *IEEE Trans. on Computers*, vol. C-37, p. 190, 1988.

[25] D. Zhang, G. A. Jullien, W. C Miller and E. E. Swartzlander, "Arithmetic for Digital Neural Networks," *Proceedings of the 10th Symposium on Computer Arithmetic*, Grenoble, June 26-28, 1991.

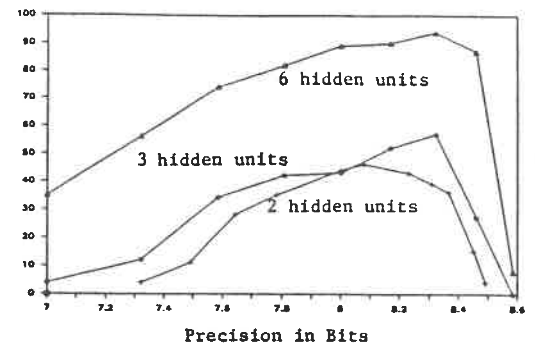

Figure 1. Number of weight sets (out of 100 initial random weight sets from [-2,2]) that converged within 0.1 using 2, 3, and 6 hidden unit XOR nets and f_2 with fixed word size $W = 13$, $M = R + F = 11$ for various values of R and F.

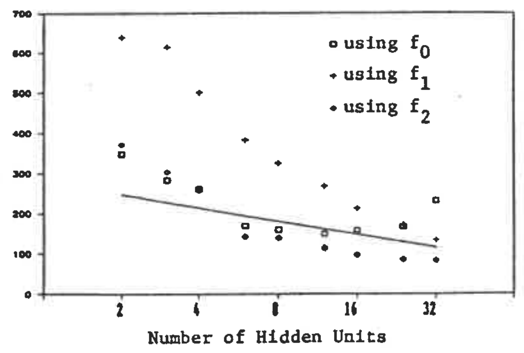

Figure 2. Mean number of epochs to converge to XOR as a function of hidden units for $M = 12, F = 9, R = 3, \eta = 2.0$. The straight line is Chauvin's prediction [18].

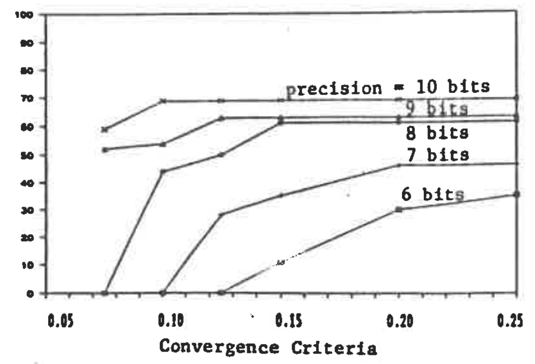

Figure 3. Number of two hidden unit XOR nets that converge as a function of precision for various convergence criteria using $f_2, R = 3$.

Proceedings Intern. AMSE Conference Neural Networks, San Diego (USA), May 29-31, 1991, Vol. 1, pp. 87-94

SUPERVISED LEARNING VIA DYNAMIC PROGRAMMING

P.Saratchandran
School of Electrical and Electronic Engineering
Nanyang Technological Institute, Nanyang Ave, Singapore 2263.

Abstract: A new algorithm for training a multilayer neural network is derived using the principles of dynamic programming. The algorithm computes the optimal values for weights on a layer after layer basis starting from the output layer of the network.The advantage of this algorithm is that it provides a minimizing error function for every hidden layer expressed entirely in terms of the weights and outputs of the hidden layer and is well suited for parallel implementation.

1 Introduction

Dynamic programming [1] is a well-known approach used in operations research and control engineering for optimization of multistage decision processes. The key concept behind dynamic programming is the principle of optimality [1] according to which the error to be minimized at every stage is split into two components viz i) the error,if any, due to the current stage and ii) the minimum error for the remaining stages obtained by using optimum decision sequences for the remainder of the process. A multilayer feed forward neural network can be thought of as a multistage decision process since optimal selection of weights for each layer is akin to optimal choice of decisions at each stage and weights in a layer can affect only the outputs of subsequent layers as with decisions in a multistage decision process.

This paper presents a learning algorithm based on dynamic programming for multilayer networks. According to this algorithm at every layer the error obtained using optimal values for weights for the remainder of the network is minimized using the weights in the current layer. Optimum weights are then computed recursively starting from the output layer. In section 2 mathematical formulation of the weight minimiza-